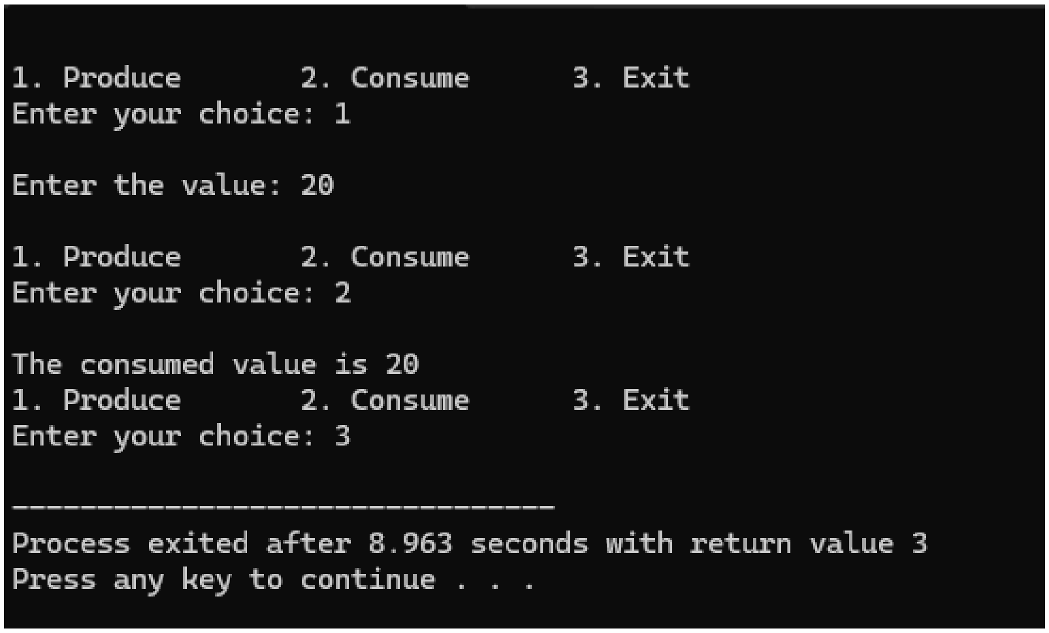
Syed Ahab Ali DT-22049 LAB 4

1) **Implement the above code and paste the screen shot of the output.**

```
#include <stdio.h>

void main()
{
    int buffer[10], bufsize, in, out, produce, consume, choice = 0;
    in = 0;
    out = 0;
    bufsize = 10;

    while (choice != 3)
    {
        printf("\n1. Produce \t 2. Consume \t 3. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            if ((in + 1) % bufsize == out)
                printf("\nBuffer is Full");
            else
            {
                printf("\nEnter the value: ");
                scanf("%d", &produce);
                buffer[in] = produce;
                in = (in + 1) % bufsize;
            }
            break;

        case 2:
            if (in == out)
                printf("\nBuffer is Empty");
            else
            {
                consume = buffer[out];
                printf("\nThe consumed value is %d", consume);
                out = (out + 1) % bufsize;
            }
            break;
        }
    }
}
```

Output:


```
1. Produce       2. Consume      3. Exit
Enter your choice: 1

Enter the value: 20

1. Produce       2. Consume      3. Exit
Enter your choice: 2

The consumed value is 20
1. Produce       2. Consume      3. Exit
Enter your choice: 3

--------------------------------
Process exited after 8.963 seconds with return value 3
Press any key to continue . . .
```

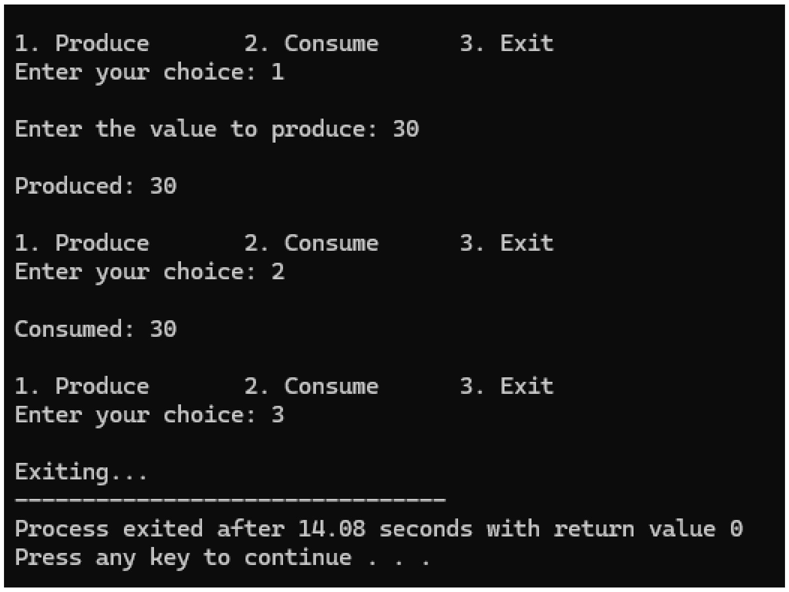
2) **Solve the producer-consumer problem using linked list. (You can perform this task using any programming language) Note: Keep the buffer size to 10 places.**

```
#include <iostream>

using namespace std;

#define BUFFER_SIZE 10 // Maximum buffer size

// Node structure for linked list
struct Node
{
  int data;
  Node *next;
};

Node *front = nullptr;
Node *rear = nullptr;
int count = 0;

void produce(int value)
{
  if (count == BUFFER_SIZE)
  {
    cout << "\nBuffer is Full";
    return;
  }
```

```
    // Creating a new node
    Node *newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    // Inserting at the end
    if (rear == nullptr)
    {
        front = rear = newNode;
    }
    else
    {
        rear->next = newNode;
        rear = newNode;
    }

    count++;
    cout << "\nProduced: " << value;
}

void consume()
{
    if (count == 0)
    {
        cout << "\nBuffer is Empty";
        return;
    }

    // Removing from front
    Node *temp = front;
    int consumedValue = temp->data;
    front = front->next;

    if (front == nullptr)
    { // If queue becomes empty
        rear = nullptr;
    }

    delete temp;
    count--;
    cout << "\nConsumed: " << consumedValue;
}

int main()
{
    int choice, value;
```

```
  while (true)
  {
    cout << "\n\n1. Produce \t 2. Consume \t 3. Exit";
    cout << "\nEnter your choice: ";
    cin >> choice;

    switch (choice)
    {
    case 1:
      cout << "\nEnter the value to produce: ";
      cin >> value;
      produce(value);
      break;

    case 2:
      consume();
      break;

    case 3:
      cout << "\nExiting...";
      return 0;

    default:
      cout << "\nInvalid choice! Try again.";
    }
  }

  return 0;
}
```

<u>Output:</u>



```
1. Produce       2. Consume      3. Exit
Enter your choice: 1

Enter the value to produce: 30

Produced: 30

1. Produce       2. Consume      3. Exit
Enter your choice: 2

Consumed: 30

1. Produce       2. Consume      3. Exit
Enter your choice: 3

Exiting...
--------------------------------
Process exited after 14.08 seconds with return value 0
Press any key to continue . . .
```

3) In producer-consumer problem what difference will it make if we utilize stack for the buffer rather than an array?

Using a stack instead of an array (queue) in the producer-consumer problem fundamentally changes the order in which items are consumed. A queue follows a FIFO (First-In-First-Out) approach, ensuring that the oldest produced item is consumed first, making it ideal for real-world buffering scenarios. However, a stack follows a LIFO (Last-In-First-Out) approach, meaning the most recently produced item is consumed first. This can be problematic in cases where order matters, as older items might never get consumed if production is continuous. While a stack-based buffer may be useful in specific applications like function calls (stack memory) or undo operations, it is generally unsuitable for producer-consumer synchronization, as it disrupts the natural flow of production and consumption.